# Foundations for Machine Learning

L. Y. Stefanus

TU Dresden, June-July 2019

Slide 02p

# Tensors in PyTorch

## Reference

- Eli Stevens and Luca Antiga. Deep Learning with PyTorch. Manning Publications, 2019/2020.
- Ian Goodfellow and Yoshua Bengio and Aaron Courville. Deep Learning. MIT Press, 2016.

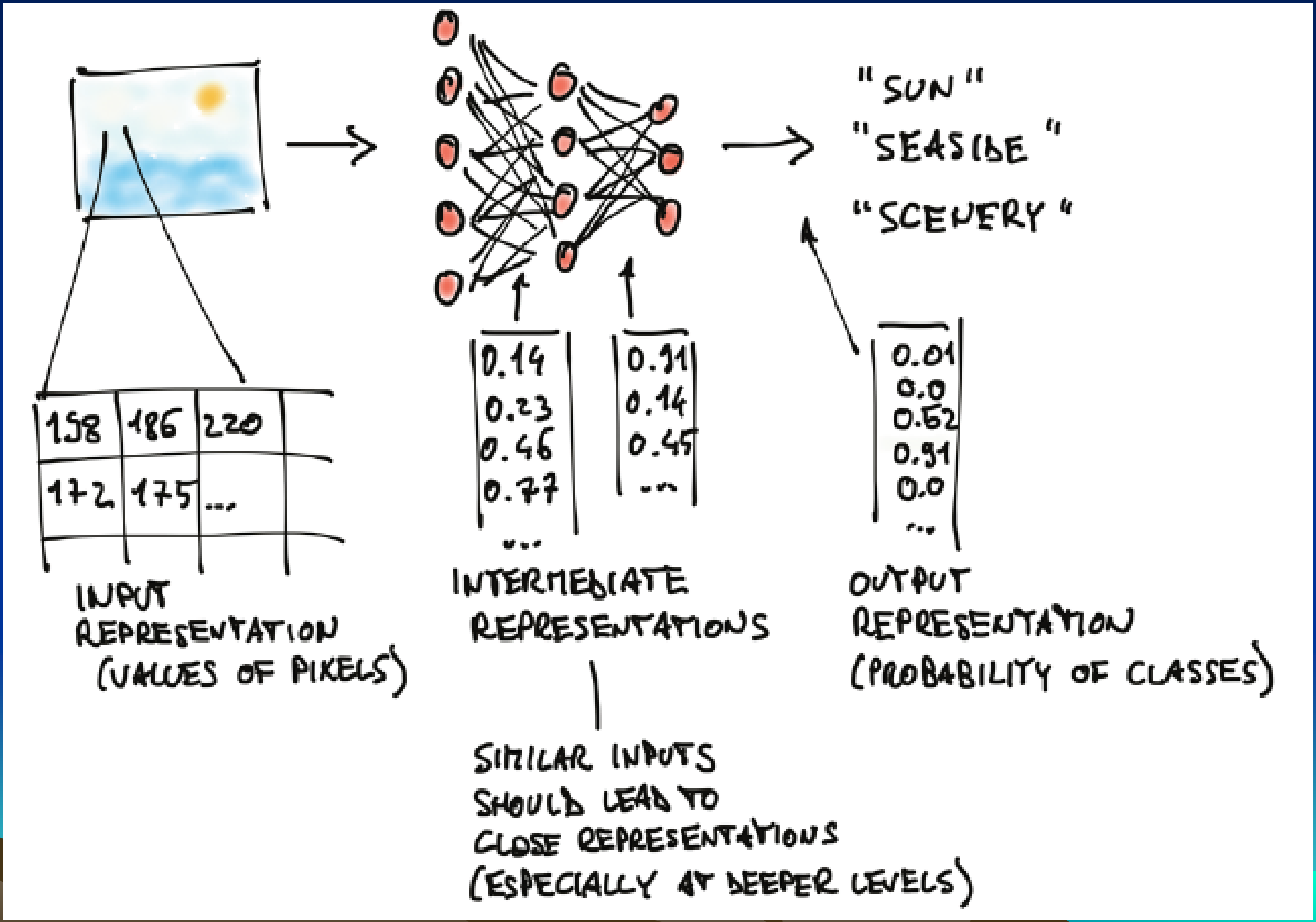
## What Is a Tensor?

- In the previous lecture we took a tour of a pre-trained neural network that can label an image according to its contents.
- This deep learning system can transform data from one representation to another, for example, from images to text labels.
- The transformation from one form of data to another is typically learned by a deep neural network in stages, which means that the partially transformed data between each stage can be thought of as a sequence of intermediate representations.

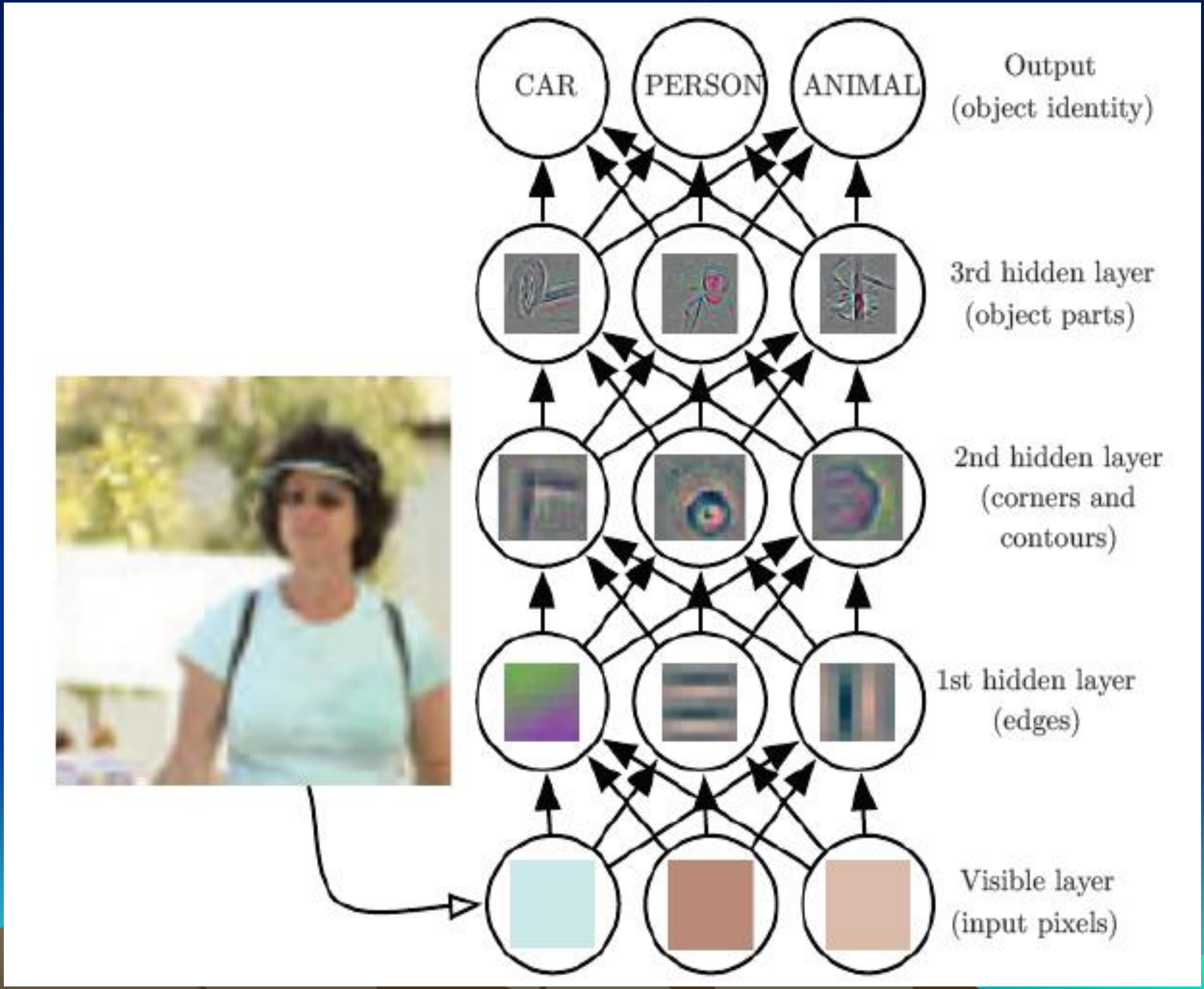
# What Is a Tensor?

- For image recognition, early representations can be things like edge detection or certain textures like fur. Deeper representations can capture more complex structures like ears, noses, or eyes.
- In general, such intermediate representations are collections of floating point numbers that characterize the input and capture the structure in the data, in a way that is instrumental for describing how inputs are mapped to the outputs of the neural network. These collections of floating point numbers and their manipulation is at the heart of modern AI.
- For these representations and their computation, PyTorch uses a fundamental data structure: **tensor**.

## Figure 3.1 A deep neural network learns how to transform an input representation to an output representation

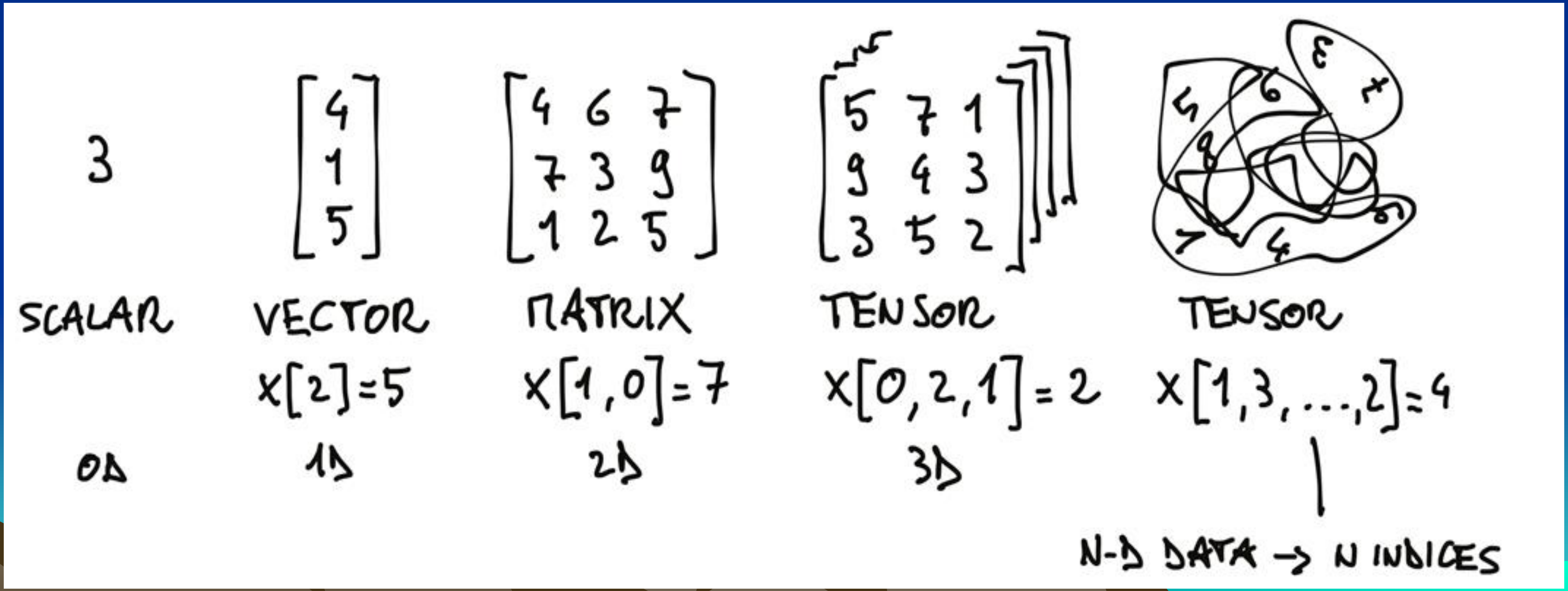

"SUN"
"SEASIDE"
"SCENERY"
158 186 220
172 175 ...
0.14 0.23 0.46 0.77 ...
0.91 0.14 0.45 ...
0.01 0.0 0.52 0.91 0.0 ...
INPUT REPRESENTATION (VALUES OF PIXELS)
INTERMEDIATE REPRESENTATIONS
OUTPUT REPRESENTATION (PROBABILITY OF CLASSES)
SIMILAR INPUTS SHOULD LEAD TO CLOSE REPRESENTATIONS (ESPECIALLY AT DEEPER LEVELS)

# A Deep Learning Model



# What Is a Tensor?

- In Machine Learning, tensors refer to the generalization of vectors and matrices to an arbitrary number of dimensions, similar to multidimensional arrays.
- The dimensionality of a tensor coincides with the number of indices used to refer to scalar values within the tensor.



## What Is a Tensor?

- PyTorch is not the only library dealing with multi-dimensional arrays. NumPy is by far the most popular multidimensional array library in Python.
- PyTorch shows seamless interoperability with NumPy, which brings with it, integration with the rest of the scientific libraries in Python, such as SciPy, Scikit-learn, and Pandas.

# What Is a Tensor?

- Compared to NumPy arrays, PyTorch tensors have a few superpowers, such as the ability to perform very fast operations on Graphical Processing Units, to distribute operations on multiple devices or machines, or to keep track of the graph of computations that created them.
- These are all important features when implementing a modern deep learning system.
- Understanding the capabilities and API of tensors is important in order to have a strong implementation tool for machine learning.

# Fundamentals of Tensors

- A tensor is a data structure, like an array, capable of storing collection of numbers that are accessible individually using an index, and that can be indexed with multiple indices.
- Firstly, let’s see list indexing so we can compare it to tensor indexing. Take a list of three numbers in Python:

```
In[1]: a = [1.0, 2.0, 1.0]
In[2]: a[0]
Out[2]:
1.0
In[3]: a[2] = 3.0
       a
# Out[3]:
[1.0, 2.0, 3.0]
```

# Fundamentals of Tensors

- It is not unusual for simple Python programs dealing with vectors of numbers, such as the coordinates of a 2D line, to use Python lists to store the vector.
- This can be sub-optimal for several reasons:

1. *Numbers in Python are full-fledged objects.* While a floating point number might take only, for instance, 32 bits to be represented on a computer, Python will box them in a full-fledged Python object with reference counting, etc.; this is not a problem if we need to store a small number of them, but allocating millions of such numbers gets very inefficient;

# Fundamentals of Tensors

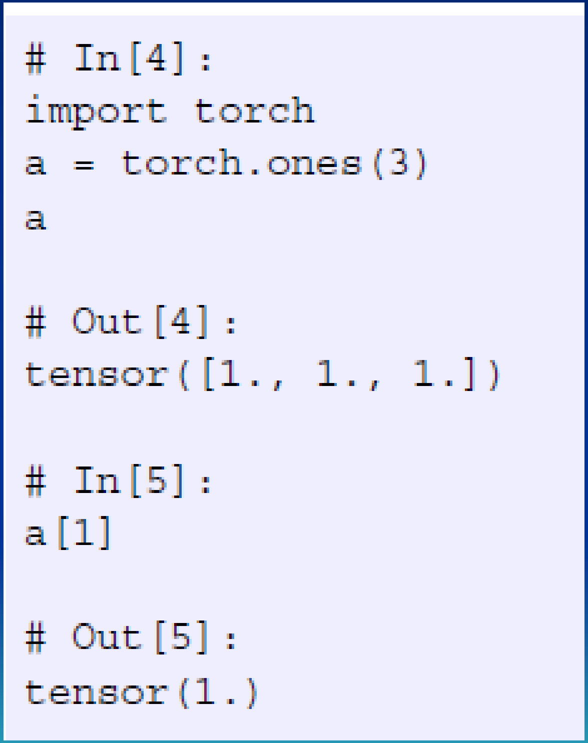
2. *Lists in Python are meant for sequential collections of objects.* There are no operations defined for, say, efficiently taking the dot product of two vectors, or summing vectors together; also, Python lists have no way of optimizing the layout of their content in memory, as they are indexable collections of pointers to Python objects (of any kind, not just numbers); last, Python lists are 1D, and while one can create lists of lists, this is again very inefficient;
3. *The Python interpreter is slow compared to executing optimized compiled code.* Performing mathematical operations on large collections of numerical data can be much faster using optimized code written in a compiled, low-level language like C.

## Fundamentals of Tensors

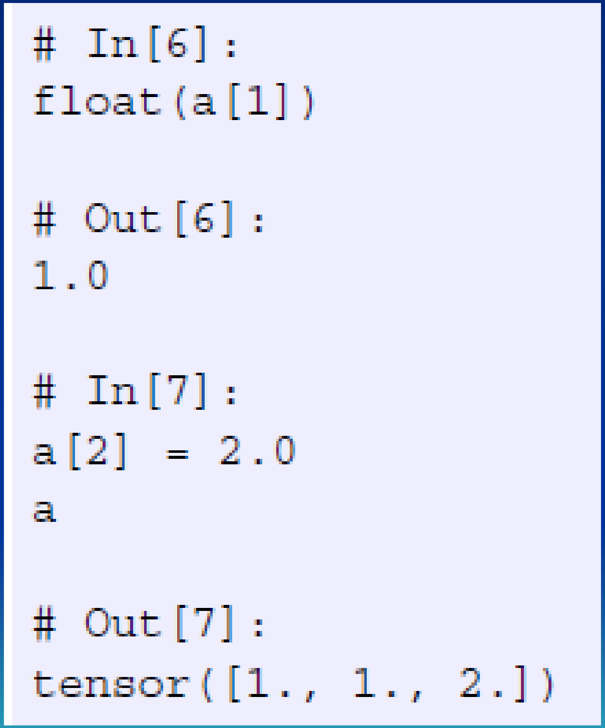
- For these reasons, data science libraries rely on NumPy, or introduce dedicated data structures like PyTorch tensors, that provide efficient low-level implementations of numerical data structures and related operations on them, wrapped in a convenient high-level API.
- We will learn later that many types of data, from images to time series, audio, even sentences, can be represented using tensors. By defining operations over tensors, we can slice and manipulate data expressively and efficiently, even from a high-level (and not particularly fast) language such as Python.

# Fundamentals of Tensors

- Let’s construct a simple PyTorch tensor, consisting of just three ones in a row.



```
# In[4]:
import torch
a = torch.ones(3)
a

# Out[4]:
tensor([1., 1., 1.])

# In[5]:
a[1]

# Out[5]:
tensor(1.)
```



```
# In[6]:
float(a[1])

# Out[6]:
1.0

# In[7]:
a[2] = 2.0
a

# Out[7]:
tensor([1., 1., 2.])
```

- Let’s see what we did here: after importing the torch module, we called a function that creates a 1D tensor of size 3 filled with the value 1.0. We can access an element using its 0-based index or assign a new value to it.
- Although on the surface the example above doesn’t differ all that much from a list of number objects, under the hood things are completely different. Python lists or tuples of numbers are collections of Python objects that are individually allocated in memory, as shown in Figure-3.3.

- PyTorch Tensors on the other hand are views over (typically) contiguous memory blocks containing *unboxed* C numeric types, (4 bytes) float in this case, as we can see on the right side of Figure-3.3.
- This means that a 1D tensor of 1,000,000 float numbers will require exactly 4,000,000 contiguous bytes to be stored, plus a small overhead for the meta data (e.g. dimensions, numeric type).

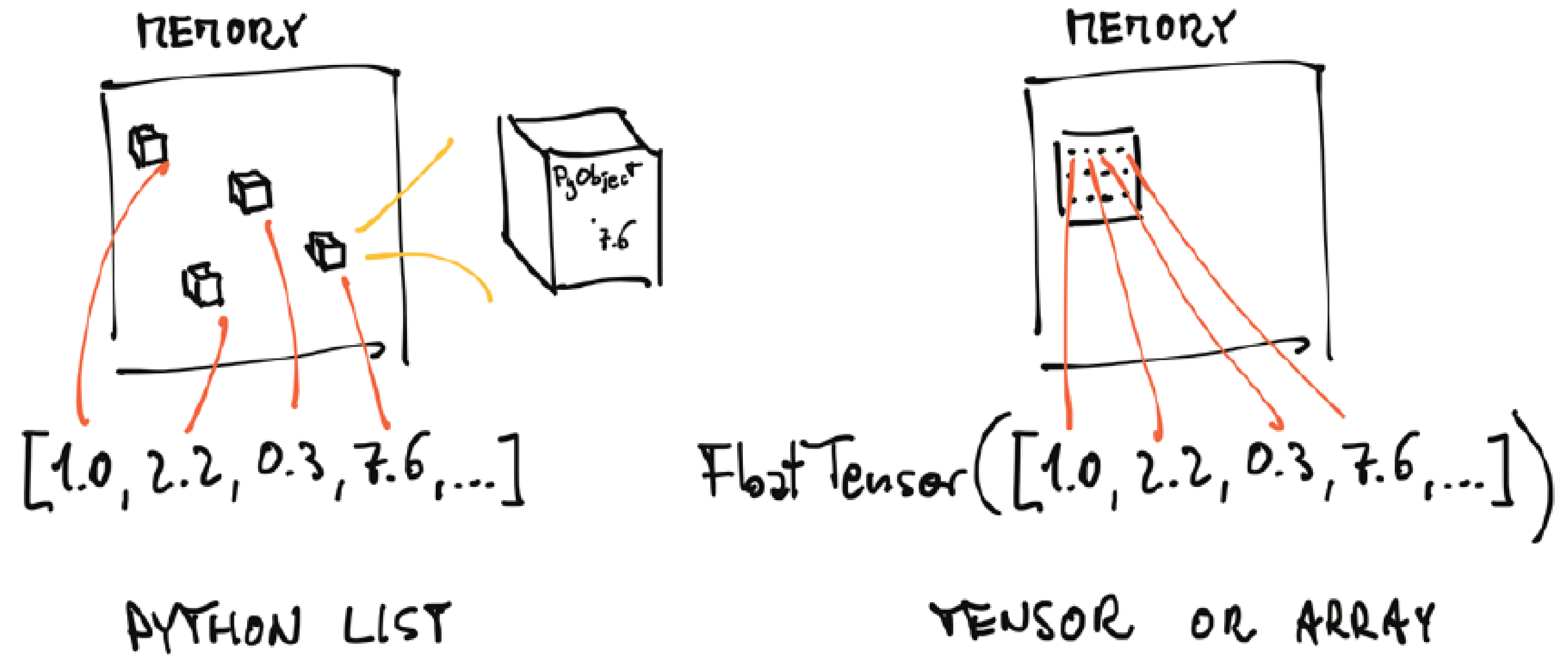
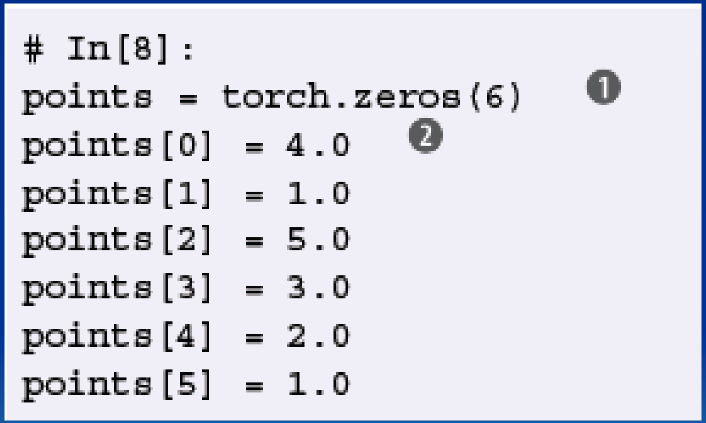


Figure 3.3 Python object (boxed) numeric values vs. tensor (unboxed array) numeric values.

# Fundamentals of Tensors

- Say we have a list of 2D coordinates we'd like to represent a geometrical object, let's say a triangle. Instead of having coordinates as numbers in a Python list, we can use a 1D tensor, by storing x's in the even indices and y's in the odd indices, like:



```
# In[8]:
points = torch.zeros(6)   ❶
points[0] = 4.0    ❷
points[1] = 1.0
points[2] = 5.0
points[3] = 3.0
points[4] = 2.0
points[5] = 1.0
```

(1) The use of zeros here is just a way to get an appropriately sized tensor.

(2) We overwrite those zeros with the values we actually want.

# Fundamentals of Tensors

- We can also pass a Python list to the constructor:

```
# In[11]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points

# Out[11]:
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])
```

- Here we passed a list of lists to the constructor. We can ask the tensor about its shape, which informs us on the size of the tensor along each dimension.

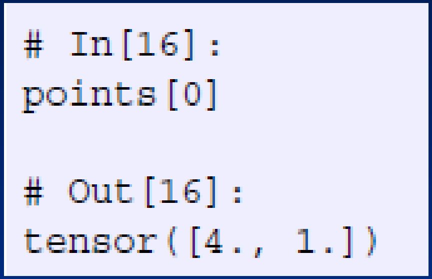
```
# In[12]:
points.shape

# Out[12]:
torch.Size([3, 2])
```

- We can access an individual element in the tensor using two indices now, for instance:

```
# In[14]:
points = torch.FloatTensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points

# Out[14]:
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])

# In[15]:
points[0, 1]

# Out[15]:
tensor(1.)
```

returns the y-coordinate of the 0-th point in our dataset.

- We can also access the first element in the tensor to get the 2D coordinates of the first point.



```
# In[16]:
points[0]

# Out[16]:
tensor([4., 1.])
```

- Note that what we get as the output is another tensor, only a 1D tensor of size 2 containing the values in the first row of the points tensor.
- Does it mean that a new chunk of memory was allocated, values were copied into it, and the new memory returned wrapped in a new tensor object? No, because that would be very inefficient, especially if we had millions of points. What we got back is instead a different **view** of the same underlying data, limited to the first row.

## Tensors and Storages

- Let’s look deeper at the implementation under the hood: Values are allocated in contiguous chunks of memory, managed by torch.Storage instances.
- A storage is a one-dimensional array of numerical data, i.e. a contiguous block of memory containing numbers of a given type, such a float or int32. A PyTorch Tensor is a view over such a Storage that is capable of indexing into that storage using an offset and per-dimension strides.

# Tensors and Storages

- Multiple tensors can index the same storage, even if they index into the data differently. See an example in Figure-3.4.
- In fact, when we requested points[0] in the last snippet, what we got back is another tensor that indexes the same storage as the points tensor, just not all of it and with different dimensionality (1D vs 2D).
- The underlying memory is only allocated once, however, so creating alternate tensor-views on the data can be done quickly, no matter the size of the data managed by the Storage instance.

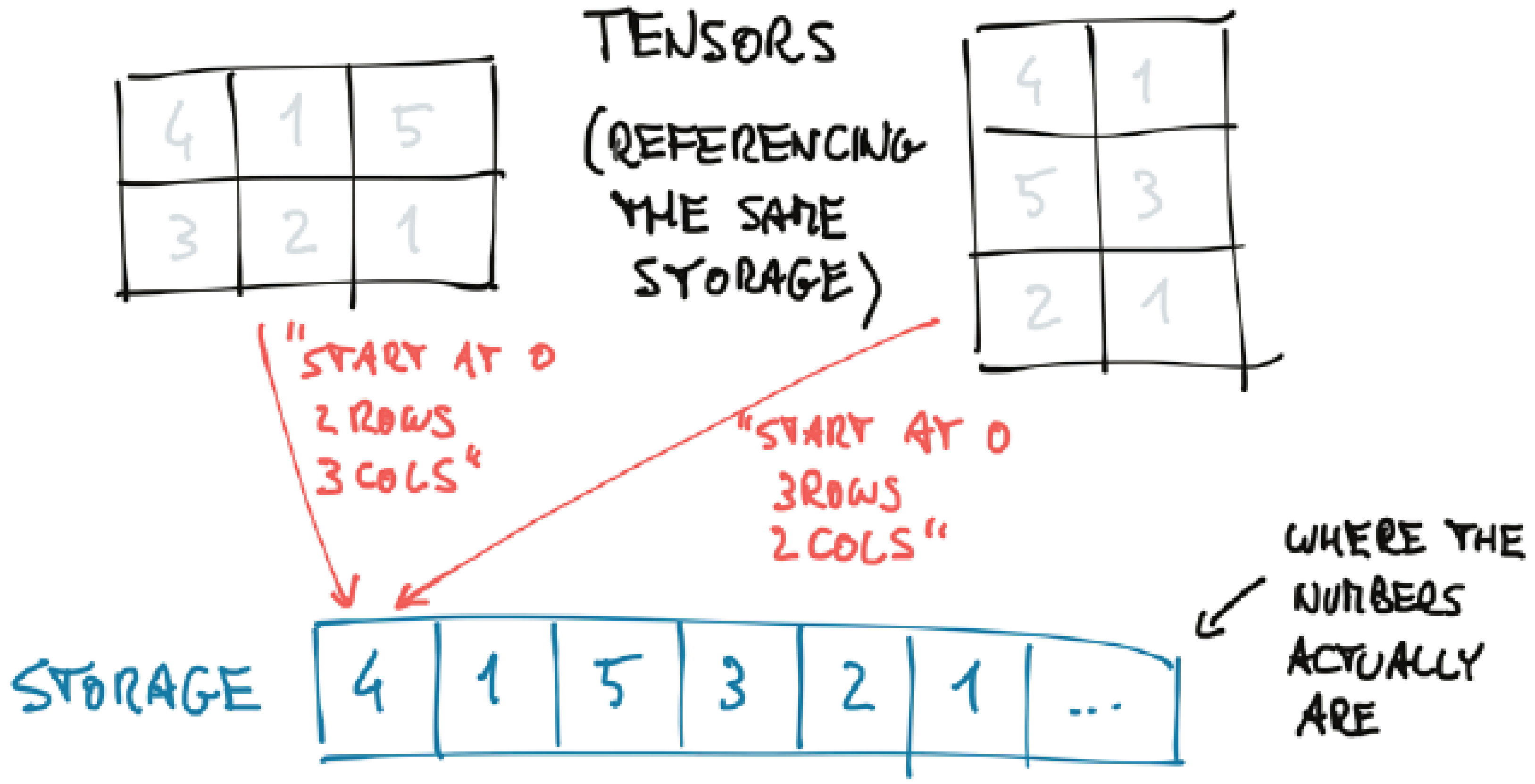
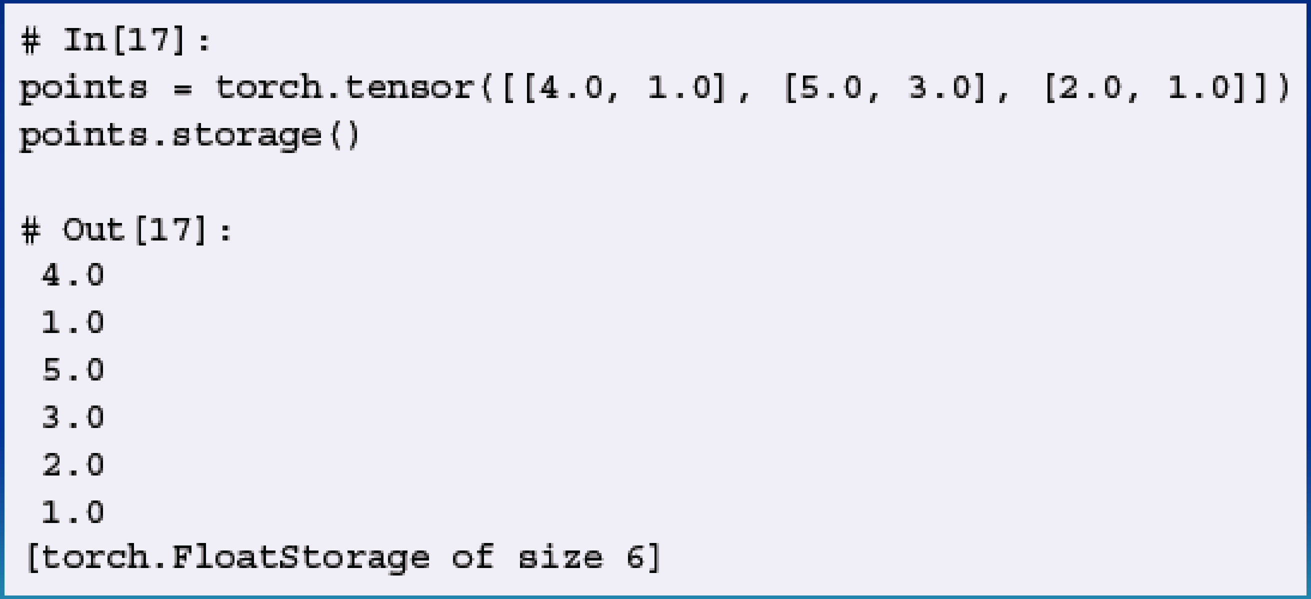


Figure 3.4 Tensors are views over a Storage instance.

## Tensors and Storages

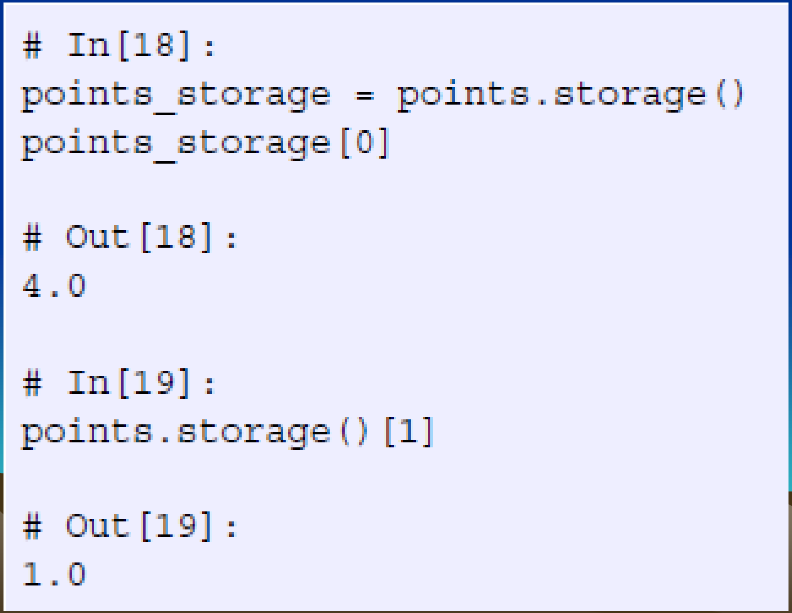
- Let’s see how indexing into the storage works in practice with our 2D points. The storage for a given tensor is accessible using the .storage property:



```
# In[17]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points.storage()

# Out[17]:
 4.0
 1.0
 5.0
 3.0
 2.0
 1.0
[torch.FloatStorage of size 6]
```

## Tensors and Storages

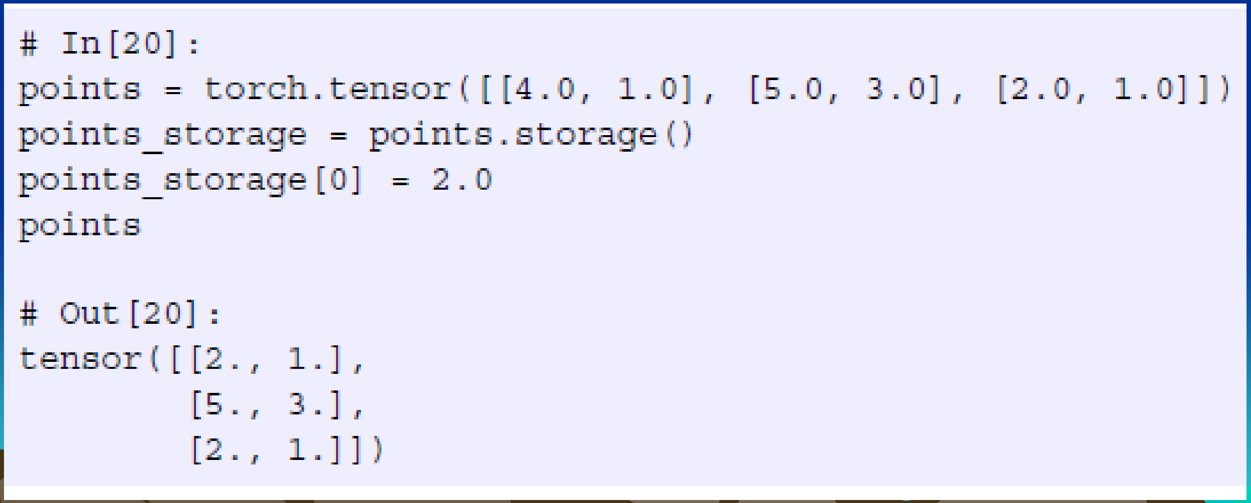
- Even though the tensor reports itself as having 3 rows and 2 columns, the storage under the hood is a contiguous array of size 6. In this sense, the tensor just knows how to translate a pair of indices into a location in the storage.
- We can also index into a storage manually, for instance:



```
# In[18]:
points_storage = points.storage()
points_storage[0]

# Out[18]:
4.0

# In[19]:
points.storage()[1]

# Out[19]:
1.0
```

# Tensors and Storages

- We can not index a storage of a 2D tensor using two indices. The layout of a storage is always one-dimensional, irrespective of the dimensionality of any and all tensors that might refer to it.
- Changing the value of a storage leads to changing the content of its referring tensor:



```
# In[20]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points_storage = points.storage()
points_storage[0] = 2.0
points

# Out[20]:
tensor([[2., 1.],
        [5., 3.],
        [2., 1.]])
```

# Tensors and Storages

- We will seldom, if ever, use storage instances directly, but understanding the relationship between a tensor and the underlying storage is very useful to understand the cost (or lack thereof) of certain operations later on.
- It’s a good mental model to keep in mind when we want to write effective PyTorch code.

# Size, offset, stride

- In order to index into a storage, tensors rely on a few pieces of information, which, together with their storage, define them: size, storage offset and stride. See Figure-3.5.
- The size (or shape, in NumPy parlance) is a tuple indicating how many elements across each dimension the tensor represents.
- The storage offset is the index in the storage corresponding to the first element in the tensor.
- The stride is the number of elements in the storage that need to be skipped over to obtain the next element along each dimension.

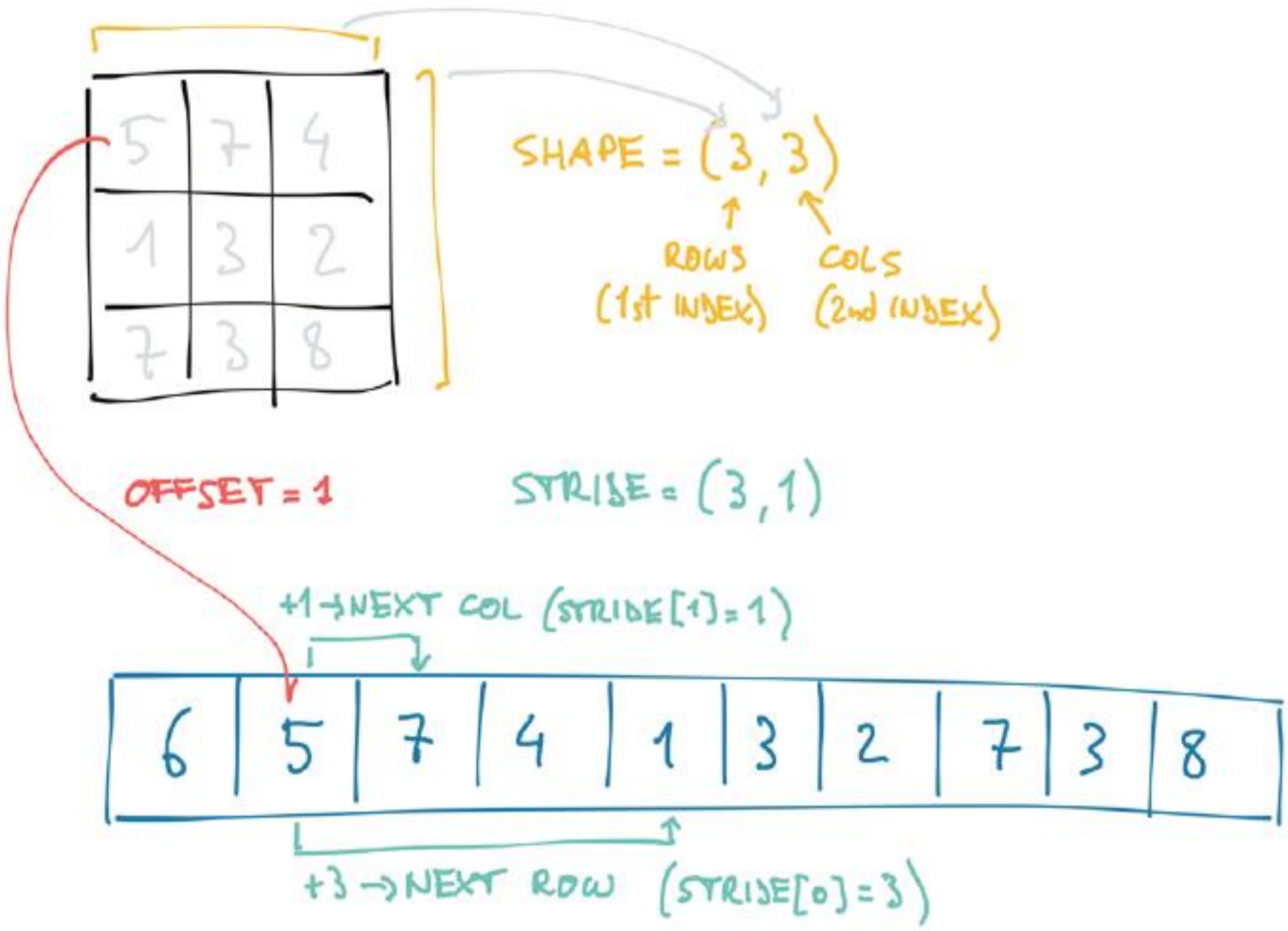
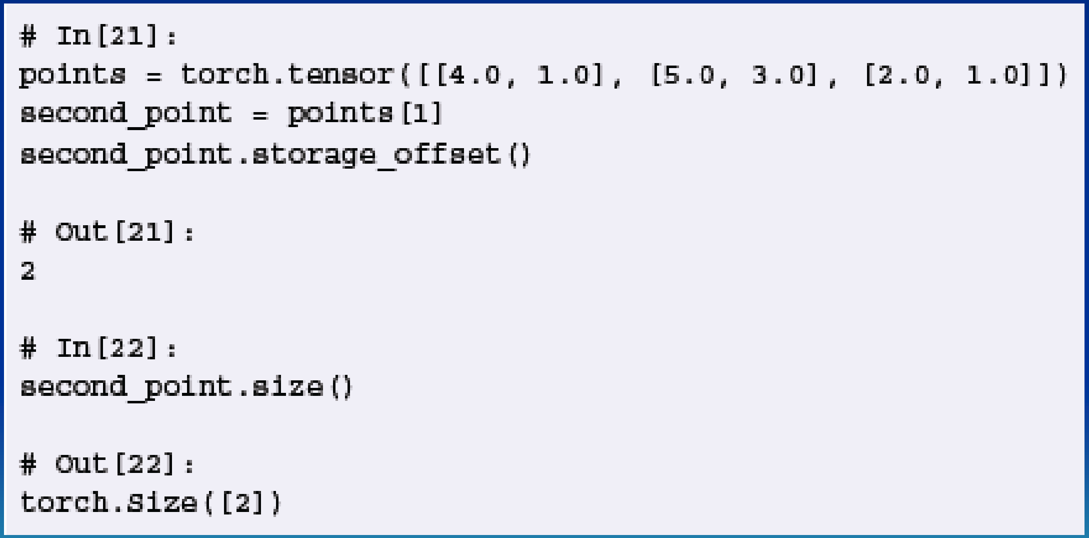


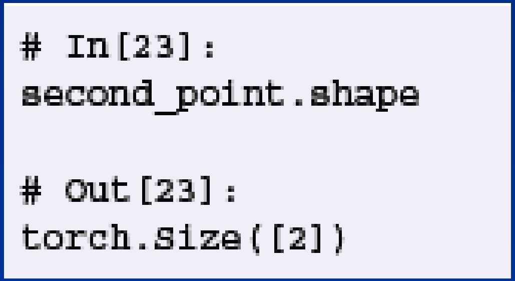
Figure 3.5 Relationship between a tensor's offset, size and stride.

## Size, offset, stride

- We can get the second point in the tensor by providing the corresponding index.
- The resulting tensor has offset 2 in the storage, since we need to skip the first point, which has two items.



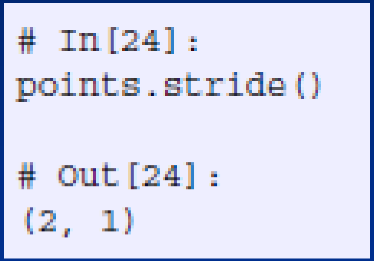
```
# In[21]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
second_point = points[1]
second_point.storage_offset()

# Out[21]:
2

# In[22]:
second_point.size()

# Out[22]:
torch.Size([2])
```



```
# In[23]:
second_point.shape

# Out[23]:
torch.Size([2])
```

# Size, offset, stride

- Last, stride is a tuple indicating the number of elements in the storage that have to be skipped when the index is increased by 1 in each dimension. For instance, our tensor points has a stride of (2, 1):



```
# In[24]:
points.stride()

# Out[24]:
(2, 1)
```

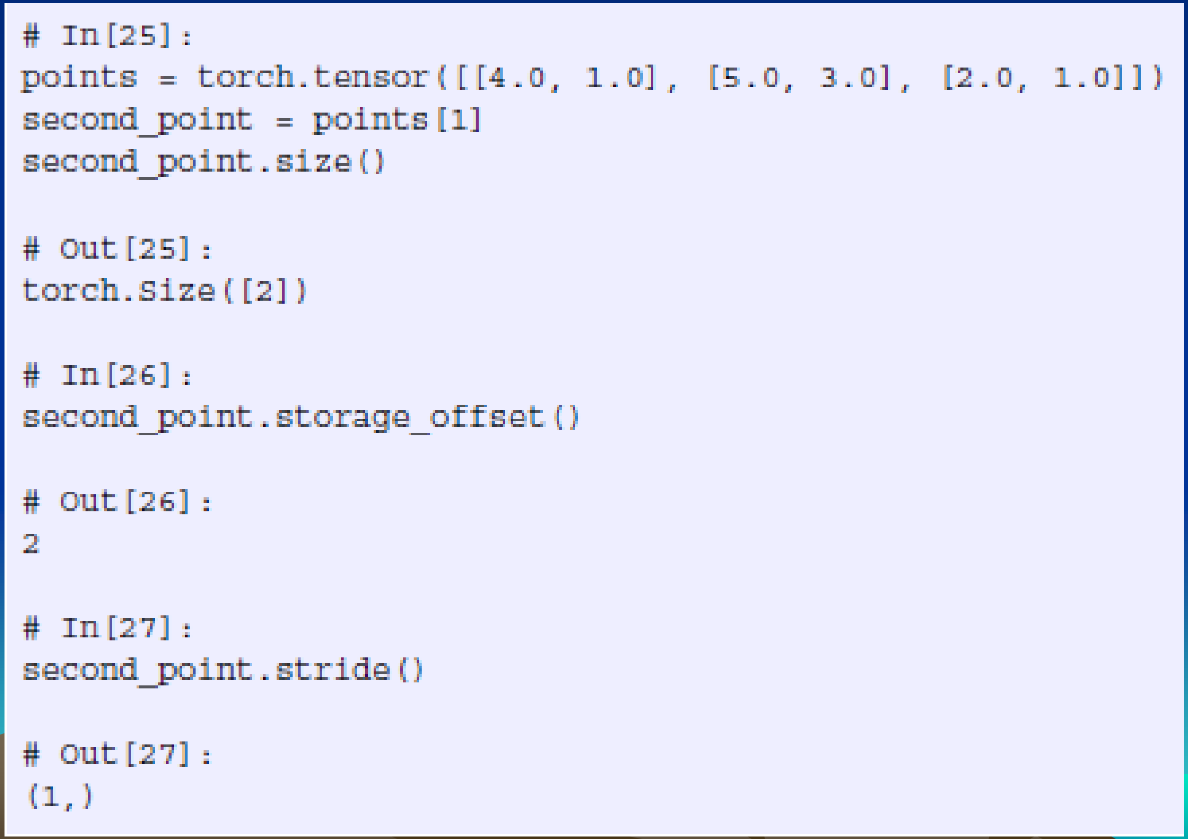
- Accessing an element i, j in a 2D tensor, results in accessing the storage_offset + stride[0] * i + stride[1] * j element in the storage.
- The offset will usually be zero; if this tensor is a view into a storage created to hold a larger tensor the offset might be a positive value.

# Size, offset, stride

- This relationship between Tensor and Storage leads some operations, like transposing a tensor or extracting a sub-tensor, to be inexpensive, as they do not lead to memory reallocations; instead they consist in allocating a new tensor object with a different value for size, storage offset or stride.

# Size, offset, stride

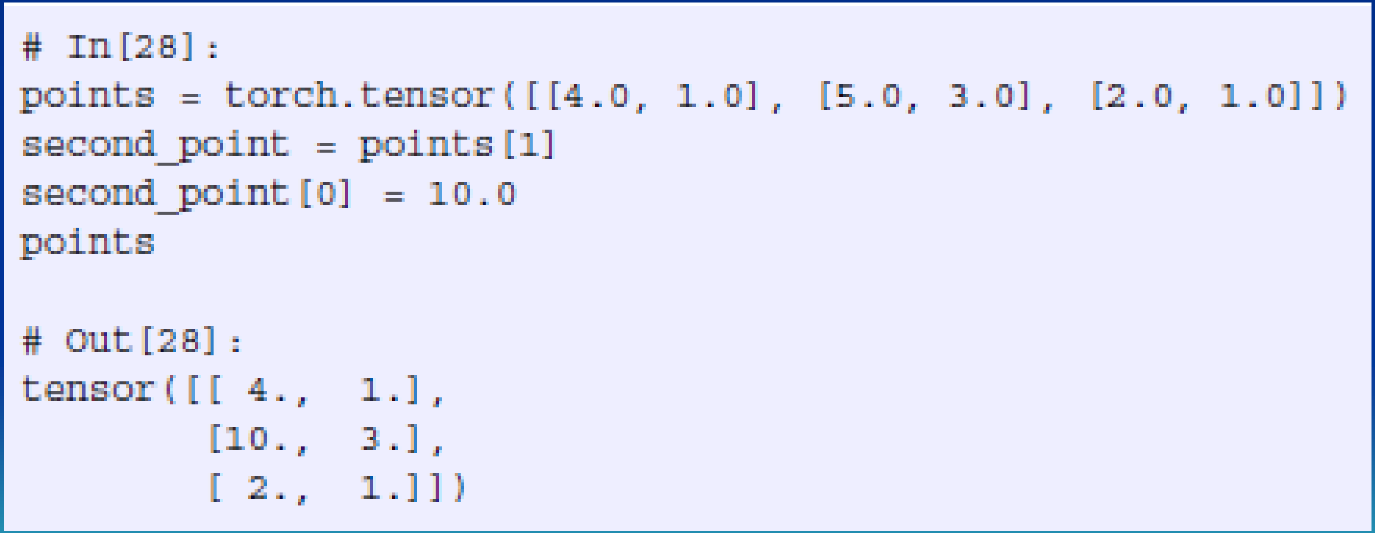
- We've already seen extracting a sub-tensor when we indexed a specific point and saw the storage offset increasing. Let's see what happens to size and stride as well:



```
# In[25]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
second_point = points[1]
second_point.size()

# Out[25]:
torch.Size([2])

# In[26]:
second_point.storage_offset()

# Out[26]:
2

# In[27]:
second_point.stride()

# Out[27]:
(1,)
```

# Size, offset, stride

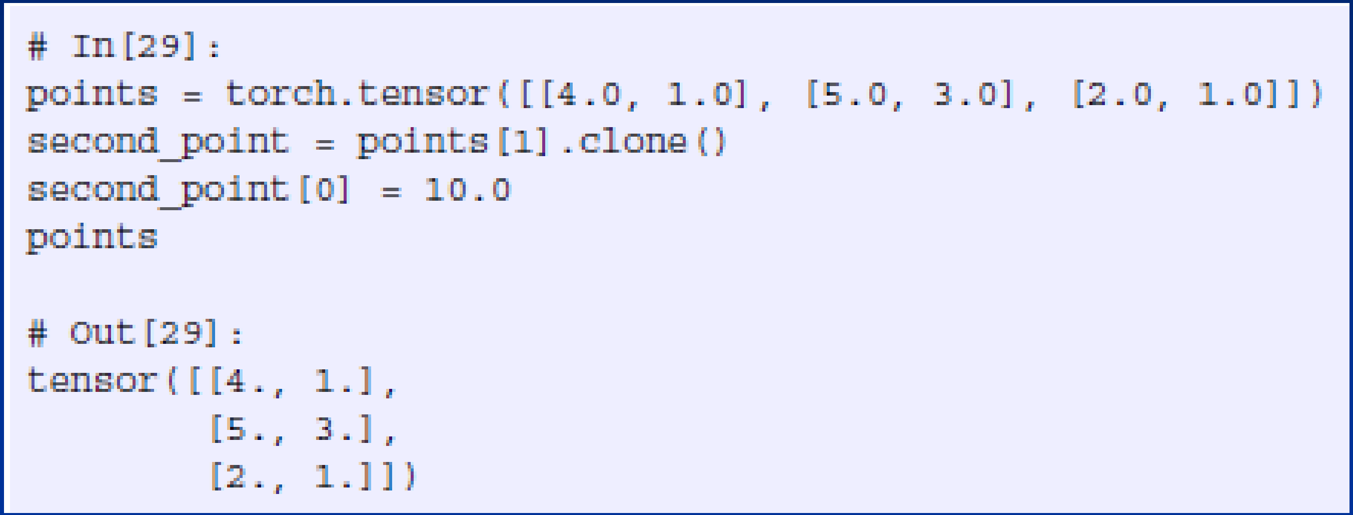
- In this case, the sub-tensor has one fewer dimension, as one would expect, while still indexing the same storage as the original points tensor. This also means that changing the sub-tensor will have a side-effect on the original tensor, too.



```
# In[28]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
second_point = points[1]
second_point[0] = 10.0
points

# Out[28]:
tensor([[ 4.,  1.],
        [10.,  3.],
        [ 2.,  1.]])
```

# Size, offset, stride

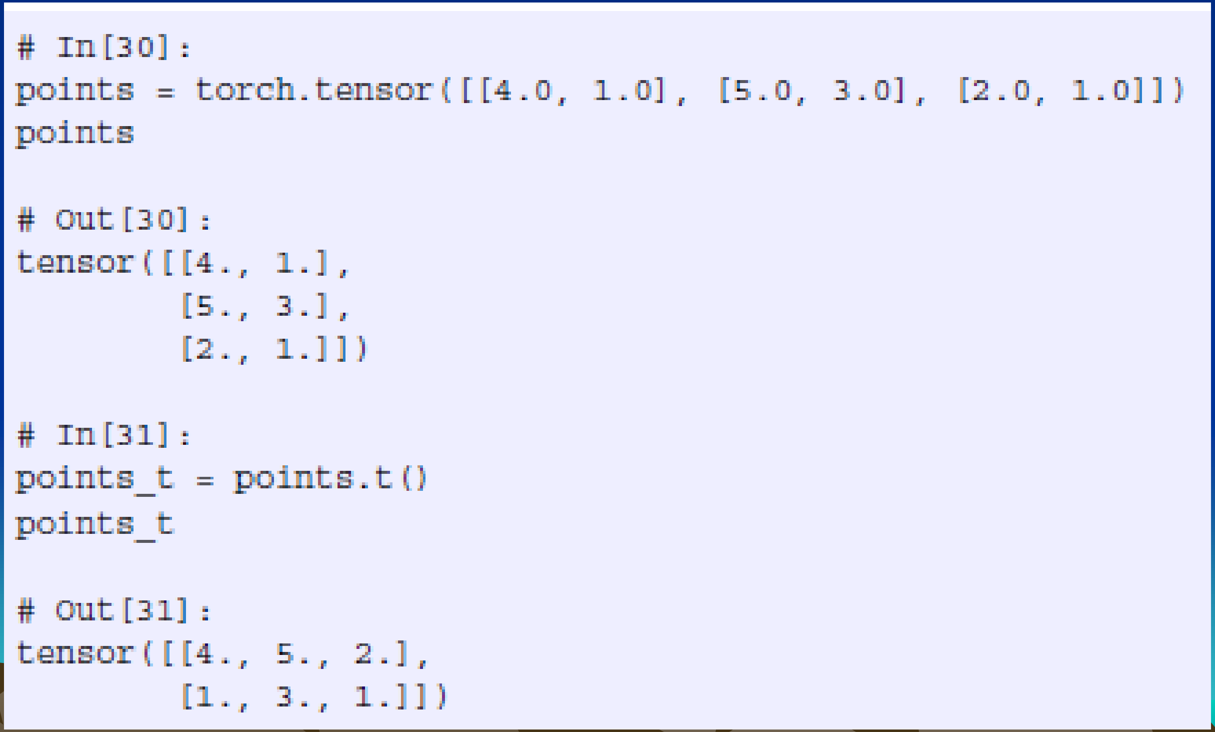
- This might not always be desirable, so we can eventually clone the sub-tensor into a new tensor.



```
# In[29]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
second_point = points[1].clone()
second_point[0] = 10.0
points

# Out[29]:
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])
```

# Size, offset, stride

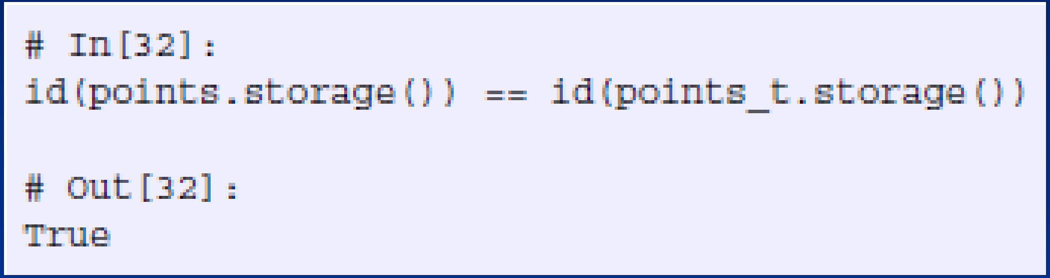
- Let’s try with transposing now. Let’s take our tensor, that has individual points points in the rows and xy coordinates in the columns, and turn it around so that individual points are along the columns.



```
# In[30]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points

# Out[30]:
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])

# In[31]:
points_t = points.t()
points_t

# Out[31]:
tensor([[4., 5., 2.],
        [1., 3., 1.]])
```

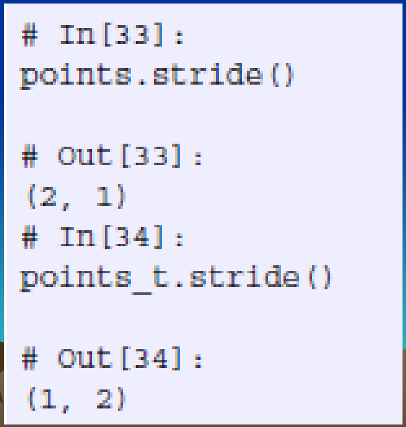
# Size, offset, stride

- We can easily verify that the two tensors share the same storage:



```
# In[32]:
id(points.storage()) == id(points_t.storage())

# Out[32]:
True
```

- and that they only differ by the shape and stride:



```
# In[33]:
points.stride()

# Out[33]:
(2, 1)
# In[34]:
points_t.stride()

# Out[34]:
(1, 2)
```

## Size, offset, stride

- The stride (2,1) above tells us that increasing the first index by one in points, e.g. going from points[0,0] to points[1,0], will skip along the storage by two elements, while increasing the second index, e.g. from points[0,0] to points[0,1] will skip along the storage by one.
- In other words, the storage holds the elements in the tensor points sequentially row by row.
- No new memory is allocated: transposing is obtained only by creating a new Tensor instance with different stride ordering from the original.

# Size, offset, stride

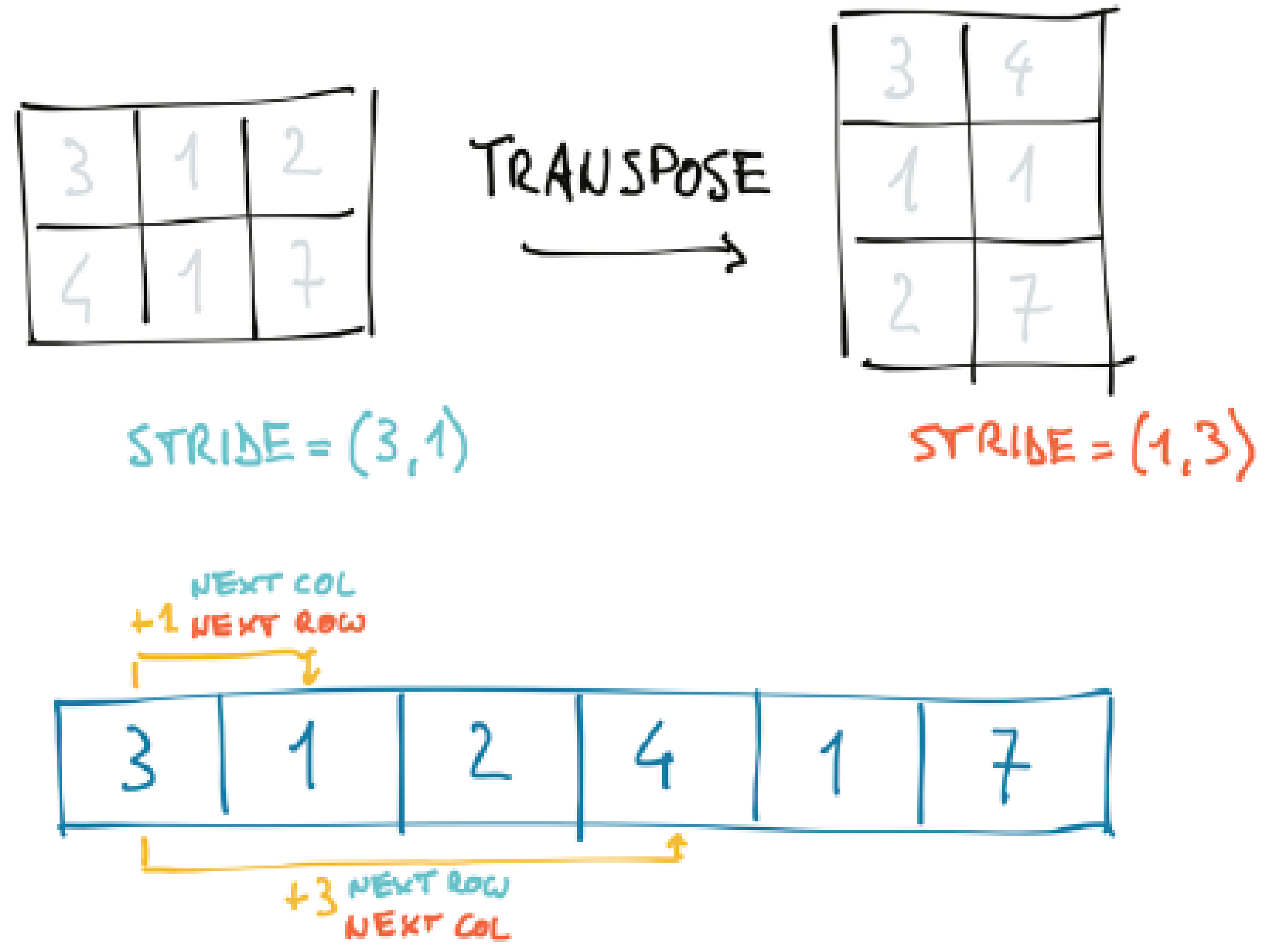


Figure 3.6 Transpose operation applied to a tensor.

# Size, offset, stride

- Transposing in PyTorch is not limited to matrices. We can transpose a multidimensional array by specifying the two dimensions along which transposing (i.e. flipping shape and stride) should occur.

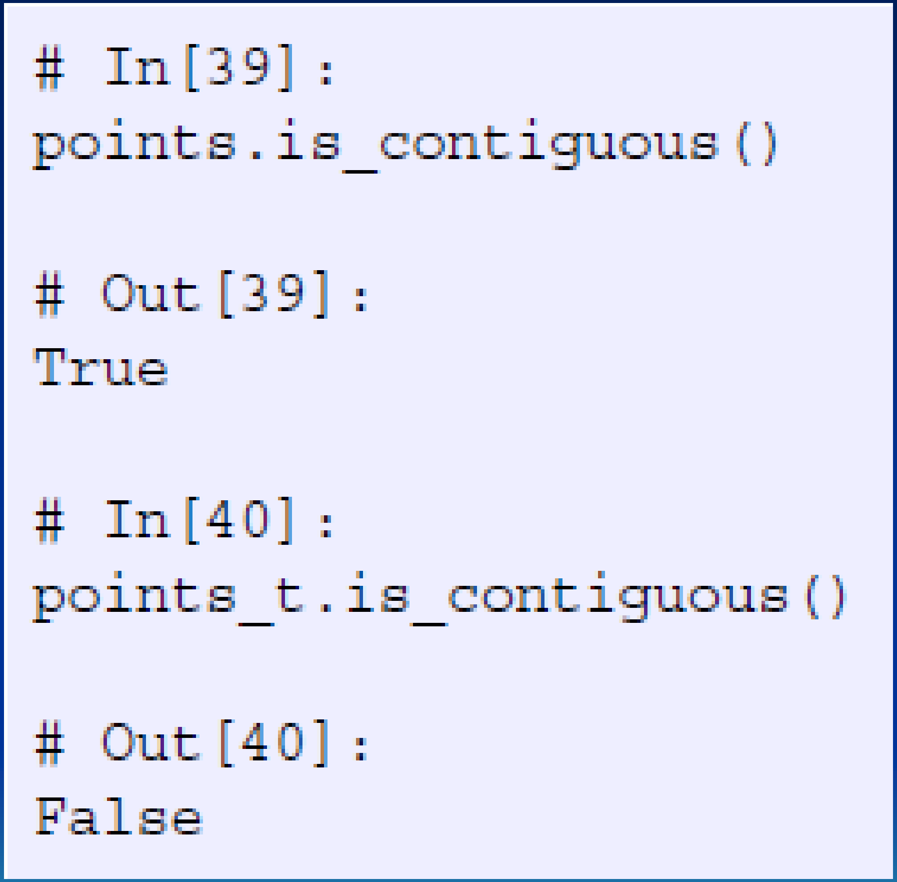
```
# In[35]:
some_tensor = torch.ones(3, 4, 5)
some_tensor_t = some_tensor.transpose(0, 2)
some_tensor.shape

# Out[35]:
torch.Size([3, 4, 5])

# In[36]:
some_tensor_t.shape

# Out[36]:
torch.Size([5, 4, 3])
```

```
# In[37]:
some_tensor.stride()

# Out[37]:
(20, 5, 1)

# In[38]:
some_tensor_t.stride()

# Out[38]:
(1, 5, 20)
```

## Size, offset, stride

- A tensor whose values are laid out in the storage starting from the right-most dimension onwards (i.e. moving along rows for a 2D tensor), is defined as contiguous.
- Contiguous tensors are convenient, because we can visit them efficiently in order without jumping around in the storage (improving data locality improves performance because of the way memory access works on modern CPUs).
- In our case, points is contiguous, while its transpose is not.

## Size, offset, stride



```
# In[39]:
points.is_contiguous()

# Out[39]:
True


# In[40]:
points_t.is_contiguous()

# Out[40]:
False
```

## Size, offset, stride

- We can obtain a new contiguous tensor from a non-contiguous one using the contiguous method. The content of the tensor will be the same, but the stride will change, as will the storage.

```
# In[41]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points_t = points.t()
points_t

# Out[41]:
tensor([[4., 5., 2.],
        [1., 3., 1.]])

# In[42]:
points_t.storage()

# Out[42]:
 4.0
 1.0
 5.0
 3.0
 2.0
 1.0
[torch.FloatStorage of size 6]

# In[43]:
points_t.stride()

# Out[43]:
(1, 2)

# In[44]:
points_t_cont = points_t.contiguous()
points_t_cont

# Out[44]:
tensor([[4., 5., 2.],
        [1., 3., 1.]])

# In[45]:
points_t_cont.stride()
```

```
# Out[45]:
(3, 1)

# In[46]:
points_t_cont.storage()

# Out[46]:
 4.0
 5.0
 2.0
 1.0
 3.0
 1.0
[torch.FloatStorage of size 6]
```

To be continued …